LEARN

labuladong 的算法小抄

团灭 LEETCODE 股票买卖问题

《labuladong 的算法秘籍》、《labuladong 的刷题笔记》和刷题插件 2.0 免费开放下载，详情见 labuladong 的刷题三件套正式发布~

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

121. 买卖股票的最佳时机（简单）

122. 买卖股票的最佳时机 II（简单）

123. 买卖股票的最佳时机 III（困难）

188. 买卖股票的最佳时机 IV（困难）

309. 最佳买卖股票时机含冷冻期（中等）

714. 买卖股票的最佳时机含手续费（中等）

————

很多读者抱怨 LeetCode 的股票系列问题奇技淫巧太多，如果面试真的遇到这类问题，基本不会想到那些巧妙的办法，怎么办？**所以本文拒绝奇技淫巧，而是稳扎稳打，只用一种通用方法解决所用问题，以不变应万变。**

这篇文章参考 英文版高赞题解 的思路，用状态机的技巧来解决，可以全部提交通过。不要觉得这个名词高大上，文学词汇而已，实际上就是 DP table，看一眼就明白了。

先随便抽出一道题，看看别人的解法：

```
int maxProfit(vector<int>& prices) {
    if(prices.empty()) return 0;
    int s1 = -prices[0], s2 = INT_MIN, s3 = INT_MIN, s4 = INT_MIN;

    for(int i = 1; i < prices.size(); ++i) {
        s1 = max(s1, -prices[i]);
        s2 = max(s2, s1 + prices[i]);
        s3 = max(s3, s2 - prices[i]);
        s4 = max(s4, s3 + prices[i]);
    }
    return max(0, s4);
}
```

能看懂吧？会做了吗？不可能的，你看不懂，这才正常。就算你勉强看懂了，下一个问题你还是做不出来。为什么别人能写出这么诡异却又高效的解法呢？因为这类问题是有框架的，但是人家不会告诉你的，因为一旦告诉你，你五分钟就学会了，该算法题就不再神秘，变得不堪一击了。

本文就来告诉你这个框架，然后带着你一道一道秒杀。这篇文章用状态机的技巧来解决，可以全部提交通过。不要觉得这个名词高大上，文学词汇而已，实际上就是 DP table，看一眼就明白了。

这 6 道题目是有共性的，我就抽出来第 4 道题目，因为这道题是一个最泛化的形式，其他的问题都是这个形式的简化，看下题目：

第一题是只进行一次交易，相当于 `k = 1`；第二题是不限交易次数，相当于 `k = +infinity`（正无穷）；第三题是只进行 2 次交易，相当于 `k = 2`；剩下两道也是不限次数，但是加了交易「冷冻期」和「手续费」的额外条件，其实就是第二题的变种，都很容易处理。

如果你还不熟悉题目，可以去 LeetCode 查看这些题目的内容，本文为了节省篇幅，就不列举这些题目的具体内容了。下面言归正传，开始解题。

一、穷举框架

首先，还是一样的思路：如何穷举？

动态规划核心套路 说过，动态规划算法本质上就是穷举「状态」，然后在「选择」中选择最优解。

那么对于这道题，我们具体到每一天，看看总共有几种可能的「状态」，再找出每个「状态」对应的「选择」。我们要穷举所有「状态」，穷举的目的是根据对应的「选择」更新状态。听起来抽象，你只要记住「状态」和「选择」两个词就行，下面实操一下就很容易明白了。

```
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 择优(选择1，选择2...)
```

比如说这个问题，**每天都有三种「选择」**：买入、卖出、无操作，我们用 `buy`, `sell`, `rest` 表示这三种选择。

但问题是，并不是每天都可以任意选择这三种选择的，因为 `sell` 必须在 `buy` 之后，`buy` 必须在 `sell` 之后。那么 `rest` 操作还应该分两种状态，一种是 `buy` 之后的 `rest`（持有了股票），一种是 `sell` 之后的 `rest`（没有持有股票）。而且别忘了，我们还有交易次数 `k` 的限制，就是说你 `buy` 还只能在 `k > 0` 的前提下操作。

很复杂对吧，不要怕，我们现在的目的只是穷举，你有再多的状态，老夫要做的就是一把梭全部列举出来。

这个问题的「状态」有三个，第一个是天数，第二个是允许交易的最大次数，第三个是当前的持有状态（即之前说的 `rest` 的状态，我们不妨用 1 表示持有，0 表示没有持有）。然后我们用一个三维数组就可以装下这几种状态的全部组合：

```
dp[i][k][0 or 1]
0 <= i <= n - 1, 1 <= k <= K
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。
此问题共 n × K × 2 种状态，全部穷举就能搞定。

for 0 <= i < n:
    for 1 <= k <= K:
        for s in {0, 1}:
            dp[i][k][s] = max(buy, sell, rest)
```

而且我们可以用自然语言描述出每一个状态的含义，比如说 `dp[3][2][1]` 的含义就是：今天是第三天，我现在手上持有着股票，至今最多进行 2 次交易。再比如 `dp[2][3][0]` 的含义：今天是第二天，我现在手上没有持有股票，至今最多进行 3 次交易。很容易理解，对吧？

我们想求的最终答案是 `dp[n - 1][K][0]`，即最后一天，最多允许 `K` 次交易，最多获得多少利润。

读者可能问为什么不是 `dp[n - 1][K][1]`？因为 `dp[n - 1][K][1]` 代表到最后一天手上还持有股票，`dp[n - 1][K][0]` 表示最后一天手上的股票已经卖出去了，很显然后者得到的利润一定大于前者。

记住如何解释「状态」，一旦你觉得哪里不好理解，把它翻译成自然语言就容易理解了。

二、状态转移框架

现在，我们完成了「状态」的穷举，我们开始思考每种「状态」有哪些「选择」，应该如何更新「状态」。

只看「持有状态」，可以画个状态转移图：

通过这个图可以很清楚地看到，每种状态（0 和 1）是如何转移而来的。根据这个图，我们来写一下状态转移方程：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
              max( 今天选择 rest,        今天选择 sell       )
```

解释：今天我没有持有股票，有两种可能，我从这两种可能中求最大利润：

1、我昨天就没有持有，且截至昨天最大交易次数限制为 `k`；然后我今天选择 `rest`，所以我今天还是没有持有，最大交易次数限制依然为 `k`。

2、我昨天持有股票，且截至昨天最大交易次数限制为 `k`；但是今天我 `sell` 了，所以我今天没有持有股票了，最大交易次数限制依然为 `k`。

```
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
              max( 今天选择 rest,         今天选择 buy         )
```

解释：今天我持有着股票，最大交易次数限制为 `k`，那么对于昨天来说，有两种可能，我从这两种可能中求最大利润：

1、我昨天就持有着股票，且截至昨天最大交易次数限制为 `k`；然后今天选择 `rest`，所以我今天还持有着股票，最大交易次数限制依然为 `k`。

2、我昨天本没有持有，且截至昨天最大交易次数限制为 `k - 1`；但今天我选择 `buy`，所以今天我就持有股票了，最大交易次数限制为 `k`。

> 这里着重提醒一下，时刻牢记「状态」的定义，`k` 的定义并不是「已进行的交易次数」，而是「最大交易次数的上限限制」。如果确定今天进行一次交易，且要保证截至今天最大交易次数上限为 `k`，那么昨天的最大交易次数上限必须是 `k - 1`。

这个解释应该很清楚了，如果 `buy`，就要从利润中减去 `prices[i]`，如果 `sell`，就要给利润增加 `prices[i]`。今天的最大利润就是这两种可能选择中较大的那个。

注意 `k` 的限制，在选择 `buy` 的时候相当于开启了一次交易，那么对于昨天来说，交易次数的上限 `k` 应该减小 1。

> 修正：以前我以为在 `sell` 的时候给 `k` 减小 1 和在 `buy` 的时候给 `k` 减小 1 是等效的，但细心的读者向我提出质疑，经过深入思考我发现前者确实是错误的，因为交易是从 `buy` 开始，如果 `buy` 的选择不改变交易次数 `k` 的约束，会出现交易次数超出限制的的错误。

现在，我们已经完成了动态规划中最困难的一步：状态转移方程。**如果之前的内容你都可以理解，那么你已经可以秒杀所有问题了，只要套这个框架就行了。**不过还差最后一点点，就是定义 base case，即最简单的情况。

```
dp[-1][...][0] = 0
解释：因为 i 是从 0 开始的，所以 i = -1 意味着还没有开始，这时候的利润当然是 0。

dp[-1][...][1] = -infinity
解释：还没开始的时候，是不可能持有股票的。
因为我们的算法要求一个最大值，所以初始值设为一个最小值，方便取最大值。

dp[...][0][0] = 0
解释：因为 k 是从 1 开始的，所以 k = 0 意味着根本不允许交易，这时候利润当然是 0。

dp[...][0][1] = -infinity
解释：不允许交易的情况下，是不可能持有股票的。
因为我们的算法要求一个最大值，所以初始值设为一个最小值，方便取最大值。
```

把上面的状态转移方程总结一下：

```
base case：
dp[-1][...][0] = dp[...][0][0] = 0
dp[-1][...][1] = dp[...][0][1] = -infinity

状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

读者可能会问，这个数组索引是 -1 怎么编程表示出来呢，负无穷怎么表示呢？这都是细节问题，有很多方法实现。现在完整的框架已经完成，下面开始具体化。

三、秒杀题目

第一题，k = 1

直接套状态转移方程，根据 base case，可以做一些化简：

```
dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])
dp[i][1][1] = max(dp[i-1][1][1], dp[i-1][0][0] - prices[i])
            = max(dp[i-1][1][1], -prices[i])
解释：k = 0 的 base case，所以 dp[i-1][0][0] = 0。

现在发现 k 都是 1，不会改变，即 k 对状态转移已经没有影响了。
可以进行进一步化简去掉所有 k：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], -prices[i])
```

直接写出代码：

```
int n = prices.length;
int[][] dp = new int[n][2];
for (int i = 0; i < n; i++) {
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
}
return dp[n - 1][0];
```

显然 `i = 0` 时 `i - 1` 是不合法的索引，这是因为我们没有对 `i` 的 base case 进行处理，可以这样给一个特化处理：

```
if (i - 1 == -1) {
    dp[i][0] = 0;
    // 根据状态转移方程可得：
    //   dp[i][0]
    // = max(dp[-1][0], dp[-1][1] + prices[i])
    // = max(0, -infinity + prices[i]) = 0

    dp[i][1] = -prices[i];
    // 根据状态转移方程可得：
    //   dp[i][1]
    // = max(dp[-1][1], dp[-1][0] - prices[i])
    // = max(-infinity, 0 - prices[i])
    // = -prices[i]
    continue;
}
```

第一题就解决了，但是这样处理 base case 很麻烦，而且注意一下状态转移方程，新状态只和相邻的一个状态有关，其实不用整个 `dp` 数组，只需要一个变量储存相邻的那个状态就足够了，这样可以把空间

复杂度降到 O(1):

```
// 原始版本
int maxProfit_k_1(int[] prices) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case
            dp[i][0] = 0;
            dp[i][1] = -prices[i];
            continue;
        }
        dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_k_1(int[] prices) {
    int n = prices.length;
    // base case: dp[-1][0] = 0, dp[-1][1] = -infinity
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        // dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        // dp[i][1] = max(dp[i-1][1], -prices[i])
        dp_i_1 = Math.max(dp_i_1, -prices[i]);
    }
    return dp_i_0;
}
```

两种方式都是一样的，不过这种编程方法简洁很多，但是如果没有前面状态转移方程的引导，是肯定看不懂的。后续的题目，你可以对比一下如何把 `dp` 数组的空间优化掉。

第二题，k = +infinity

如果 `k` 为正无穷，那么就可以认为 `k` 和 `k - 1` 是一样的。可以这样改写框架：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
            = max(dp[i-1][k][1], dp[i-1][k][0] - prices[i])

我们发现数组中的 k 已经不会改变了，也就是说不需要记录 k 这个状态了：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])
```

直接翻译成代码：

```
// 原始版本
int maxProfit_k_inf(int[] prices) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case
            dp[i][0] = 0;
            dp[i][1] = -prices[i];
            continue;
        }
        dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i-1][1], dp[i-1][0] - prices[i]);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_k_inf(int[] prices) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, temp - prices[i]);
    }
    return dp_i_0;
}
```

第三题，k = +infinity with cooldown

每次 `sell` 之后要等一天才能继续交易。只要把这个特点融入上一题的状态转移方程即可：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])
解释：第 i 天选择 buy 的时候，要从 i-2 的状态转移，而不是 i-1 。
```

翻译成代码：

```
// 原始版本
int maxProfit_with_cool(int[] prices) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case 1
            dp[i][0] = 0;
            dp[i][1] = -prices[i];
            continue;
        }
        if (i - 2 == -1) {
            // base case 2
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
            // i - 2 小于 0 时根据状态转移方程推出对应 base case
            dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
            //   dp[i][1]
            // = max(dp[i-1][1], dp[-1][0] - prices[i])
            // = max(dp[i-1][1], 0 - prices[i])
            // = max(dp[i-1][1], -prices[i])
            continue;
        }
        dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i-1][1], dp[i-2][0] - prices[i]);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_with_cool(int[] prices) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    int dp_pre_0 = 0; // 代表 dp[i-2][0]
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
```

```
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, dp_pre_0 - prices[i]);
        dp_pre_0 = temp;
    }
    return dp_i_0;
}
```

第四题，k = +infinity with fee

每次交易要支付手续费，只要把手续费从利润中减去即可。改写方程：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee)
解释：相当于买入股票的价格升高了。
在第一个式子里减也是一样的，相当于卖出股票的价格减小了。
```

> 如果直接把 `fee` 放在第一个式子里减，会有测试用例无法通过，错误原因是整型溢出而不是思路问题。一种解决方案是把代码中的 `int` 类型都改成 `long` 类型，避免 `int` 的整型溢出。

直接翻译成代码，注意状态转移方程改变后 base case 也要做出对应改变：

```
// 原始版本
int maxProfit_with_fee(int[] prices, int fee) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case
            dp[i][0] = 0;
            dp[i][1] = -prices[i] - fee;
            //   dp[i][1]
            // = max(dp[i - 1][1], dp[i - 1][0] - prices[i] - fee)
            // = max(dp[-1][1], dp[-1][0] - prices[i] - fee)
            // = max(-inf, 0 - prices[i] - fee)
            // = -prices[i] - fee
            continue;
        }
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i] - fee);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_with_fee(int[] prices, int fee) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, temp - prices[i] - fee);
    }
    return dp_i_0;
}
```

第五题，k = 2

`k = 2` 和前面题目的情况稍微不同，因为上面的情况都和 `k` 的关系不太大。要么 `k` 是正无穷，状态转移和 `k` 没关系了；要么 `k = 1`，跟 `k = 0` 这个 base case 挨得近，最后也没有存在感。

这道题 `k = 2` 和后面要讲的 `k` 是任意正整数的情况中，对 `k` 的处理就凸显出来了。我们直接写代码，边写边分析原因。

```
原始的状态转移方程，没有可化简的地方
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

按照之前的代码，我们可能想当然这样写代码（错误的）：

```
int k = 2;
int[][][] dp = new int[n][k + 1][2];
for (int i = 0; i < n; i++) {
    if (i - 1 == -1) {
        // 处理 base case
        dp[i][k][0] = 0;
        dp[i][k][1] = -prices[i];
        continue;
    }
    dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
    dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
}
return dp[n - 1][k][0];
```

为什么错误？我这不是照着状态转移方程写的吗？

还记得前面总结的「穷举框架」吗？就是说我们必须穷举所有状态。其实我们之前的解法，都在穷举所有状态，只是之前的题目中 `k` 都被化简掉了。

比如说第一题，`k = 1` 时的代码框架：

```
int n = prices.length;
int[][] dp = new int[n][2];
for (int i = 0; i < n; i++) {
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
}
return dp[n - 1][0];
```

但当 `k = 2` 时，由于没有消掉 `k` 的影响，所以必须要对 `k` 进行穷举：

```
// 原始版本
int maxProfit_k_2(int[] prices) {
    int max_k = 2, n = prices.length;
    int[][][] dp = new int[n][max_k + 1][2];
    for (int i = 0; i < n; i++) {
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) {
                // 处理 base case
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i];
                continue;
            }
            dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
            dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
        }
    }
    // 穷举了 n × max_k × 2 个状态，正确。
    return dp[n - 1][max_k][0];
}
```

> PS：这里肯定会有读者疑惑，`k` 的 base case 是 0，按理说应该从 `k = 1, k++` 这样穷举状态 `k` 才对？而且如果你真的这样从小到大遍历 `k`，提交发现也是可以的。

这个疑问很正确，因为我们前文 动态规划答疑篇 有介绍 `dp` 数组的遍历顺序是怎么确定的，主要是根据 base case，以 base case 为起点，逐步向结果靠近。

但为什么我从大到小遍历 `k` 也可以正确提交呢？因为你注意看，`dp[i][k]` 不会依赖 `dp[i][k - 1]`，而是依赖 `dp[i - 1][k - 1]`，对于 `dp[i - 1][...]`，都是已经计算出来的。所以不管你是 `k = max_k, k--`，还是 `k = 1, k++`，都是可以得出正确答案的。

那为什么我使用 `k = max_k, k--` 的方式呢？因为这样符合语义。

你买股票，初始的「状态」是什么？应该是从第 0 天开始，而且还没有进行过买卖，所以最大交易次数限制 `k` 应该是 `max_k`；而随着「状态」的推移，你会进行交易，那么交易次数上限 `k` 应该不断减少，这样一想，`k = max_k, k--` 的方式是比较合乎实际场景的。

当然，这里 `k` 取值范围比较小，所以可以不用 for 循环，直接把 k = 1 和 2 的情况全部列举出来也可以：

```java
// 状态转移方程:
// dp[i][2][0] = max(dp[i-1][2][0], dp[i-1][2][1] + prices[i])
// dp[i][2][1] = max(dp[i-1][2][1], dp[i-1][1][0] - prices[i])
// dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])
// dp[i][1][1] = max(dp[i-1][1][1], -prices[i])

// 空间复杂度优化版本
int maxProfit_k_2(int[] prices) {
    // base case
    int dp_i10 = 0, dp_i11 = Integer.MIN_VALUE;
    int dp_i20 = 0, dp_i21 = Integer.MIN_VALUE;
    for (int price : prices) {
        dp_i20 = Math.max(dp_i20, dp_i21 + price);
        dp_i21 = Math.max(dp_i21, dp_i10 - price);
        dp_i10 = Math.max(dp_i10, dp_i11 + price);
        dp_i11 = Math.max(dp_i11, -price);
    }
    return dp_i20;
}
```

有状态转移方程和含义明确的变量名指导，相信你很容易看懂。其实我们可以故弄玄虚，把上述四个变量换成 `a, b, c, d`。这样当别人看到你的代码时就会大惊失色，对你肃然起敬。

第六题，k = any integer

有了上一题 `k = 2` 的铺垫，这题应该和上一题的第一个解法没啥区别。但是出现了一个超内存的错误，原来是传入的 `k` 值会非常大，`dp` 数组太大了。现在想想，交易次数 `k` 最多有多大呢？

一次交易由买入和卖出构成，至少需要两天。所以说有效的限制 `k` 应该不超过 `n/2`，如果超过，就没有约束作用了，相当于 `k = +infinity`。这种情况是之前解决过的。

直接把之前的代码重用：

```java
int maxProfit_k_any(int max_k, int[] prices) {
    int n = prices.length;
    if (n <= 0) {
        return 0;
    }
    if (max_k > n / 2) {
        // 交易次数 k 没有限制的情况
        return maxProfit_k_inf(prices);
    }

    // base case:
    // dp[-1][...][0] = dp[...][0][0] = 0
    // dp[-1][...][1] = dp[...][0][1] = -infinity
    int[][][] dp = new int[n][max_k + 1][2];
    // k = 0 时的 base case
    for (int i = 0; i < n; i++) {
        dp[i][0][1] = Integer.MIN_VALUE;
        dp[i][0][0] = 0;
    }

    for (int i = 0; i < n; i++)
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) {
                // 处理 i = -1 时的 base case
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i];
                continue;
            }
            dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
            dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
        }
    return dp[n - 1][max_k][0];
}
```

至此，6 道题目通过一个状态转移方程全部解决。

四、最后总结

本文给大家讲了如何通过状态转移的方法解决复杂的问题，用一个状态转移方程秒杀了 6 道股票买卖问题，现在想想，其实也不算难对吧？这已经属于动态规划问题中较困难的了。

关键就在于列举出所有可能的「状态」，然后想想怎么穷举更新这些「状态」。一般用一个多维 `dp` 数组储存这些状态，从 base case 开始向后推进，推进到最后的状态，就是我们想要的答案。想想这个过程，你是不是有点理解「动态规划」这个名词的意义了呢？

具体到股票买卖问题，我们发现了三个状态，使用了一个三维数组，无非还是穷举 + 更新，不过我们可以说的高大上一点，这叫「三维 DP」，怕不怕？这个大实话一说，立刻显得你高人一等，名利双收有没有，所以给个在看/分享吧，鼓励一下我。

————————————————

《labuladong 的算法小抄》已经出版，关注公众号「labuladong」查看详情；后台回复关键词「进群」可加入算法群，回复题号获取对应的文章：